Chapter TWO
Inheritance and Polymorphism

Exam Objectives

Implement inheritance including visibility modifiers and composition.
Override hashCode, equals, and toString methods from Object class.
Implement polymorphism.
Develop code that uses abstract classes and methods.

Inheritance

At the core of an object-oriented language, there's the concept of inheritance.

In simple terms, inheritance refers to an **IS-A** relationship where a class (called superclass) provides common attributes and methods to derived or more specialized classes (called subclass).

In Java, a class is only allowed to inherit from a single superclass (singular inheritance). Of course, the only exception is `java.lang.Object`, which has no superclass. This class is the superclass of all classes.

The keyword `extends` is used to specify this relationship. For example, a hammer **IS-A** tool, so we can model this as:

```
class Tool {
    public int size;
}

class Hammer extends Tool {

}
```

As `size` is a `public` attribute, it's inherited by `Hammer`:

```
Hammer hammer = new Hammer();
hammer.size = 10;
```

From the previous chapter, we know that only `private` and members with default visibility when the subclass is defined in a different package than the superclass, are not inherited.

An attribute or method is inherited with the same visibility level as the one defined in the superclass. However, in the case of methods, you can change them to be more visible, but you cannot make them less visible:

```
class Tool {
    public int size;
    public int getSize() { return size; }
}

class Hammer extends Tool {
    private int size; // No problem!
    // Compile-time error
    private int getSize() { return size; }
}
```

There's no problem for attribute because we're creating a **NEW** attribute in Hammer that **HIDES** the one inherited from `Tool` when the name is the same.

Here are the things you can do in a subclass:

- Inherited attributes can be used directly, just like any other.
- An attribute can be declared in the subclass with the same name as the one in the superclass, thus hiding it.
- New attributes that are not in the superclass can be declared in the subclass.
- Inherited methods can be directly used as they are.
- A new instance method can be declared in the subclass that has the same signature as the one in the superclass, thus overriding it.
- A new `static` method can be declared in the subclass that has the same signature as the one in the superclass, thus hiding it.
- New methods that are not in the superclass can be declared in the subclass.
- A constructor can be declared in the subclass that invokes the constructor of the superclass, either implicitly or by using the keyword `super`.

So for methods, reducing their visibility is not allowed because they are handled in a different way, in other words, methods are either overridden or overloaded.

Besides, think about it. Because of encapsulation, attributes are supposed to be hidden, but with methods, if a subclass doesn't have a method of the superclass, the subclass cannot be used wherever the superclass is used. This is called the *Liskov substitution principle*, which is important in polymorphism, and we'll review after talking about overridden and overloaded methods.

Implementing an interface is in some ways is a type of inheritance because they have some common characteristics, but by doing it, the relationship becomes **HAS-A**. We'll talk more about them in Chapter 4.

Overloading and Overriding

The difference between overloading and overriding has to do a lot with method signatures.

In a few words, the *method signature* is the name of the method and the list of its parameters (types and number of parameters included). Note that return types are not included in this definition.

We talk about overloading when a method changes the method signature, by changing the list of parameters of another method (that might be inherited) while keeping the same name.

```
class Hotel {
    public void reserveRoom(int rooms) { ... }
}

class ThreeStarHotel extends Hotel {
    // Method overload #1
    public void reserveRoom(List<Room> rooms) {
        ...
    }
    // Method overload #2
    public void reserveRoom(
            int rooms, int numberPeople) {
        ...
    }
}
```

Changing just the return type will generate a compile error:

```
class ThreeStarHotel extends Hotel {
    // Compile-time error, reserveRoom is seen as duplicated
    public void reserveRoom(List<Room> rooms) {
        ...
    }
    public boolean reserveRoom(List<Room> rooms) {
        ...
    }
}
```

Exceptions in the `throws` clause are not considered when overloading, so again, changing just the exception list will throw a compile error:

```
class ThreeStarHotel extends Hotel {
    // Compile-time error, reserveRoom is seen as duplicated
    public void reserveRoom(List<Room> rooms)
        throws RuntimeException {
        ...
    }
    public boolean reserveRoom(List<Room> rooms)
        throws NullPointerException {
        ...
    }
}
```

When an overloaded method is called, the compiler has to decide which version of the method is going to call. The first obvious candidate is to call the method that exactly matches the number and types of the arguments. But what happens when there isn't an exact match?

The rule to remember is that Java will look for the **CLOSEST** match **FIRST** (this means a larger type, a superclass, an autoboxed type, or the **MORE** particular type).

For example, when this class is executed:

```
class Print {
    static void printType(short param) {
        System.out.println("short");
    }
    static void printType(long param) {
        System.out.println("long");
    }
    static void printType(Integer param) {
        System.out.println("Integer");
    }
```

```
    static void printType(CharSequence param) {
       System.out.println("CharSequence");
    }

    public static void main(String[] args) {
        byte b = 1;
        int i = 1;
        Integer integer = 1;
        String s = "1";

        printType(b);
        printType(i);
        printType(integer);
        printType(s);
   }
}
```

The output is:

```
short
long
Integer
CharSequence
```

In the first method call, the argument type is `byte`. There's no method taking a `byte`, so the closest larger type is `short`.

In the second method call, the argument type is `int`. There's no method taking a `byte`, so the closest larger type is `long` (note that this has higher precedence than `Integer`).

In the third method call, the argument type is `Integer`. There's a method that takes an `Integer`, so this is called.

In the last method call, the argument type is `String`. There's no method taking a `String`, so the closest superclass is `CharSequence`.

If it can't find a match or if the compiler cannot decide because the call is ambiguous, a compile error is thrown. For example, considering the previous class, the following will cause an error because there isn't a larger type than `double` and it can't be autoboxed to an `Integer`:

```
// Can't find a match
double d = 1.0;
printType(d);
```

The following is an example of an ambiguous call, assuming the methods:

```
static void printType(float param,
                      double param2) {
    System.out.println("float-double");
}
static void printType(double param,
                      float param2) {
    System.out.println("double-float");
}

...

// Ambiguous call
printType(1,1);
```

Constructors of a class can also be overloaded. In fact, you can call one constructor from another with the `this` keyword:

```
class Print {
    Print() {
        this("Calling with default argument");
    }
    Print(String s) {
        System.out.println(s);
    }
}
```

We talk about overriding when the method signature is the same, but for some reason, we want to redefine an **INSTANCE** method in the subclass.

```
class Hotel {
    public void reserveRoom(int rooms) {
        ...
    }
}
class ThreeStarHotel extends Hotel {
    // Method override
    public void reserveRoom(int rooms) {
        ...
    }
}
```

If a `static` method with the same signature as a `static` method in the superclass is defined in the subclass, then the method is **HIDDEN** instead of overridden.

There are some rules when overriding a method.

The access modifier must be the same or with more visibility:

```
class Hotel {
    public void reserveRoom(int rooms) {
        ...
    }
}
class ThreeStarHotel extends Hotel {
    // Compile-time error
    protected void reserveRoom(int rooms) {
        ...
    }
}
```

The return type must be the same or a subtype:

```
class Hotel {
    public Integer reserveRoom(int rooms) {
        ...
    }
}

class ThreeStarHotel extends Hotel {
    // Compile-time error
    public Number reserveRoom(int rooms) {
        ...
    }
}
```

Exceptions in the `throws` clause must be the same, less, or subclasses of those exceptions:

```
class Hotel {
    public void reserveRoom(int rooms)
           throws IOException {
        ...
    }
}

class ThreeStarHotel extends Hotel {
    // Compile-time error
    public void reserveRoom(int rooms) throws Exception {
        ...
    }
}
```

Overriding is a critical concept in polymorphism, but before touching this topic, let's see some important methods from `java.lang.Object` that most of the time we'll need to override.

Object class methods

In Java, all objects inherit from `java.lang.Object`.

This class has the following methods that can be overridden (redefined):

- `protected Object clone() throws CloneNotSupportedException`
- `protected void finalize() throws Throwable`
- `public int hashCode()`
- `public boolean equals(Object obj)`
- `public String toString()`

The most significant methods, the ones you almost always would want to redefine, are `hashCode`, `equals`, and `toString`.

public int hashCode()
It returns a hash code value for the object. The returned value must have the following contract:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are not equal according to the equals(`java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

public boolean equals(Object obj)
Indicates whether another object is equal to the object that calls the method. It's necessary to override the `hashCode` method whenever this method is overridden since the contract for the `hashCode` method states that equal objects must have equal hash codes. This method is:

- reflexive: for any non- `null` reference value `x`, `x.equals(x)` should return `true`.

- symmetric: for any non- `null` reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- transitive: for any non- `null` reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- consistent: for any non- `null` reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or `false`, provided no information used in equals comparisons on the objects is modified.

For any non- `null` reference value `x`, `x.equals(null)` should return `false`.

public String toString()
It returns a string representation of the object. The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object.

To override these methods just follow the general rules for overriding:

- The access modifier must be the same or more accessible
- The return type must be either the same or a subclass
- The name must be the same
- The argument list types must be the same
- The same exceptions or their subclasses are allowed to be thrown

In a few words, define the method just as it appears in the `java.lang.Object` class.

Polymorphism

Polymorphism is the ability for an object to vary its behavior based on its type. This is best demonstrated with an example:

```
class HumanBeing {
    public void dress() {
        System.out.println("Dressing a human being");
    }
}
class Man extends HumanBeing {
    public void dress() {
        System.out.println("Put on a shirt");
        System.out.println("Put on some jeans");
    }
}
class Woman extends HumanBeing {
    public void dress() {
        System.out.println("Put on a dress");
    }
}
class Baby extends HumanBeing {
    public void dress() {
        System.out.println(
            "I don't know how to dress!");
    }
}
```

And now let's create some human beings to see polymorphism in action:

```
HumanBeing[] someHumans = new HumanBeing[3];
someHumans[0] = new Man();
someHumans[1] = new Woman();
someHumans[2] = new Baby();
```

```
for(int i = 0; i < someHumans.length; i++) {
    someHumans[i].dress();
    System.out.println();
}
```

The output:

```
Put on a shirt
Put on some jeans
Put on a dress
I don't know how to dress!
```

Even though `HumanBeing` is used, the JVM decides at runtime which method to call based on the type of the object assigned, not the variable's reference type.

This is called *virtual method invocation*, a fancy name for overriding.

Overriding is also known as *dynamic polymorphism* because the type of the object is decided at **RUN** time.

In contrast, overloading is also called *static polymorphism* because it's resolved at **COMPILE** time.

Abstract classes and methods

If we examine the previous example, I think we'll agree that the implementation of the `dress()` method in the class `HumanBeing` doesn't sound exactly right.

Most of the time, we'll be working with something more concrete, like a `Man` or a `Woman` so there's no need to instantiate the `HumanBeing` class directly, however, a common abstraction of those classes may be useful. Using an abstract class (or method) is the best option to model these cases.

Abstract classes **CANNOT** be instantiated, only subclassed. They are declared with the `abstract` keyword:

```
abstract class AClass { }
```

Abstract methods are declared **WITHOUT** an implementation (body), like this:

```
abstract void AMethod();
```

So in the previous example, it's better to model the whole `HumanBeing` class as `abstract` so no one can use directly:

```
abstract class HumanBeing {
    public abstract void dress();
}
```

Now, the following would cause a compile error:

```
HumanBeing human = new HumanBeing();
```

And it makes sense; there can't be no guarantees that an `abstract` class will have all its methods implemented. Calling an unimplemented method would be an epic fail.

Here are the rules when working with `abstract` methods and classes:

The `abstract` keyword can only be applied to classes or non-static methods.

```
abstract class AClass {
    // Compile-time error
    public static abstract void AMethod();
}
```

An `abstract` class doesn't need to declare abstract methods to be declared abstract.

```
abstract class AClass { } // No problem
```

If a class includes `abstract` methods, then the class itself must be declared abstract.

```
class AClass { // Compile-time error
    public abstract void AMethod();
}
```

If the subclass of an `abstract` class doesn't provide an implementation for all `abstract` methods, the subclass must also be declared `abstract`.

```
// Compile-time error
class Man extends HumanBeing { }
```

Methods of an interface are considered abstract, so an `abstract` class that implements an interface can implement some or none of the interface methods.

```
// No problem
abstract class AClass implements Runnable {}
```

Key Points

- Inheritance refers to an I**S-A** relationship where a class (called superclass) provides common attributes and methods to derived or more specialized classes (called subclass).
- Here are the things you can do in a subclass:
  - Inherited attributes can be used directly, just like any other.
  - An attribute can be declared in the subclass with the same name as the one in the superclass, thus hiding it.
  - New attributes that are not in the superclass can be declared in the subclass.
  - Inherited methods can be used directly as they are.
  - A new instance method can be declared in the subclass that has the same signature as the one in the superclass, thus overriding it.
  - A new `static` method can be declared in the subclass that has the same signature as the one in the superclass, thus hiding it.
  - New methods that are not in the superclass can be declared in the subclass.
  - A constructor can be declared in the subclass that invokes the constructor of the superclass, either implicitly or by using the keyword `super`.
- The method signature is the name of the method and the list of its parameters (types and number of parameters included). Return types are not included in this definition.
- We talk about overloading when a method changes the list of parameters of another method (that might be inherited) while keeping the same name.
- We talk about overriding when the method signature is the same, but for some reason, we want to redefine an **INSTANCE** method in the subclass.

- The most important methods of java.lang.Object that most classes must redefine are:
  - `public int hashCode()`
  - `public boolean equals(Object obj)`
  - `public String toString()`
- With polymorphism, subclasses can define their own behaviors (different than the ones of the methods of the superclass), and the JVM will call the appropriate method for the object. This behavior is referred to as virtual method invocation.
- Abstract classes **CANNOT** be instantiated, only subclassed. Abstract methods are declared **WITHOUT** an implementation (body).
- The `abstract` keyword can only be applied to classes or non-static methods.
- An `abstract` class doesn't need to declare `abstract` methods to be declared `abstract`.
- If a class includes `abstract` methods, then the class itself must be declared `abstract`.
- If the subclass of an `abstract` class doesn't provide an implementation for all `abstract` methods, the subclass must also be declared `abstract`.
- Methods of an interface are considered `abstract`, so an `abstract` class that implements an interface can implement some or none of the interface methods.

Self Test

1. Given:

```
public class Question_2_1 {
    protected int id;
    protected String name;

    protected boolean equals(Question_2_1 q) {
        return this.name.equals(q.name);
    }

    public static void main(String[] args) {
        Question_2_1 q1 = new Question_2_1();
        Question_2_1 q2 = new Question_2_1();
        q1.name = "q1";
        q2.name = "q1";

        if(q1.equals((Object)q2)) {
            System.out.println("true");
        } else {
            System.out.println("false");
        }
    }
}
```

What is the result?
A. `true`
B. `false`
C. Compilation fails
D. An exception occurs at runtime

2. Which of the following is a method of `java.lang.Object` that can be overridden?
A. `public String toString(Object obj)`
B. `public int equals(Object obj)`
C. `public int hashCode(Object obj)`
D. `public int hashCode()`

3. Given:

```
public class Question_2_3 {
    public static void print(Integer i) {
        System.out.println("Integer");
    }
    public static void print(Object o) {
        System.out.println("Object");
    }
    public static void main(String[] args) {
        print(null);
    }
}
```

What is the result?
A. `Integer`
B. `Object`
C. Compilation fails
D. An exception occurs at runtime

4. Given:

```
class SuperClass {
    public static void print() {
        System.out.println("Superclass");
    }
}
public class Question_2_4 extends SuperClass {
    public static void print() {
        System.out.println("Subclass");
    }
    public static void main(String[] args) {
        print();
    }
}
```

What is the result?
A. Superclass
B. Subclass
C. Compilation fails
D. An exception occurs at runtime

5. Given:

```
abstract class SuperClass2 {
    public static void print() {
        System.out.println("Superclass");
    }
}
class SubClass extends SuperClass2 {}
public class Question_2_5 extends SuperClass {
    public static void main(String[] args) {
        SubClass subclass = new SubClass();
        subclass.print();
    }
}
```

What is the result?
A. `Superclass`
B. Compilation fails because an `abstract` class cannot have `static` methods
C. Compilation fails because `Subclass` doesn't implement method `print()`
D. Compilation fails because `Subclass` doesn't have a method `print()`
E. An exception occurs at runtime

6. Given:

```
abstract class SuperClass3 {
    public void print() {
        System.out.println("Superclass");
    }
}
public class Question_2_6 extends SuperClass3 {
    public void print() {
        System.out.println("Subclass");
    }
    public static void main(String[] args) {
        Question_2_6 q = new Question_2_6();
        ((SuperClass3)q).print();
    }
}
```

What is the result?
A. `Superclass`
B. `Subclass`
C. Compilation fails
D. An exception occurs at runtime

Open answers page

---

01. Encapsulation and Immutable Classes

03. Inner Classes